Evaluation of Coverage-Driven Random Verification

Qing Qin
Charles L. Brown Department of Electrical and Computer Engineering
University of Virginia
Charlottesville, United States
qq3za@virginia.edu

***Abstract*—The project focuses on examining the advantages of random verification with real examples. Random verification has two necessary parts, hierarchical testbench and coverage metrics. The layered testbench allows a verifier to improve the coverage by only modifying the randomization constraints at the highest level of abstraction. Although it might take a verifier more time to build such hierarchical testbench than a directed one, the overall verification time will be less and the process will be less error-prone. The project also evaluates two coverage metrics for monitoring the verification process, functional coverage and code coverage. The definitions of the two metrics are the same for both directed and random verification, but the ways to measure the coverage data differ for the two methods. Especially for random verification, effectively using functional coverage measurements requires a verifier to think beyond the scope of directed testing. At last, the project looks into the concept of RTL fault coverage in order to establish the relationship between functional verification and manufacturing test.**

Keywords—random verification; functional coverage; code coverage; RTL fault coverage

I. INTRODUCTION

Random verificaiton is an alternative to the traditional directed verification. In directed testing, a verifier first identifies the test cases that target some specific scenarios in the verification plan, then forces the input patterns to the device-under-test (DUT), and finally verifies if the outputs match the expected value. If the results are consistent, the DUT is proved to function correctly under the targeted scenarios. In random verification, in contrast, the verifier identifies which scenarios are covered after the testbench randomly generates some input patterns and drives the random stimuli to the DUT. Without constraints on input patterns, random verification has potential advantages in reaching corner scenarios that a verifier dismisses and in saving time and avoiding human errors from constantly modifying the whole testbench as in directed verification. The goal of the project is to examine the advantages of random verification with real examples and to evalueate the verification method as the alternative to direct testing.

The major characteristics of random verification for evaluation include hierachical testbhench and different coverage metrics In addition, the project also looks into the concept of RTL fault coverage to seek the relatinoship between functional verification and manufacturing test. To gaurantee progress and yield solid results, the project uses some questions as a guildeline and demonstrates the resuilts using a 64-bit floating-point adder and a RAM as DUT. The questions include how to create a layered testbench that can be shared by different tests, how to model the behavior of DUT to predict the outputs with random inputs, what are the definitions of the coverage metrics, and how to use the coverage measurements to mornitor the verification process.

The rest of the paper is divided into three parts to present the answers to the questions above. Section II presentes the layered testbench used throughout the project. Section III introduces two coverage metrics for evalutation. Section IV shows the method to collect RTL fault coverage and the results on the floating-point adder. Section V concludes the paper.

II. HIERARCHICAL TESTBENCH

Fig. 1 shows the architecture of the layered testbench used for verifying the RAM and the floating-point adder in the project. The testbench design is adapted from [1] and implmented in SystemVerilog. The generator at the scenario layer generates random inputs under a set of constraints. The verifier can constantly change the constraints to improve the number of scenarios covered by the generated set of random stimuli. Modifying the testbench only at this highest level of abstraction saves time and avoids error as compared to modify the whole testbench in the case of direct verification. The driver at the command layer takes the test cases from the generator and applies the signals at appropriate timing to DUT. The scoreboard at the functional layer models the behavior of the DUT at a higher abstraction level than RTL.

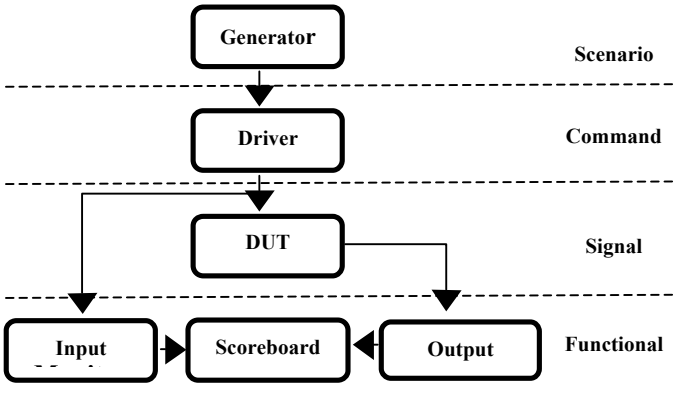

Fig. 1. Architecture of Layered Testbench Used in Two Demos

Applying the same hierarchical testbench to both the RAM and the adder shows another advantage in random verification over directed testing that an object-oriented layered testbench has reusable verification classes, such as monitors, driver, and generator. Although the scoreboard component has high dependencies on designs and takes most of efforts for a verifier to build in some cases, modeling in other high-level languages, such as Matlab and C, can help improve the efficiency, There is also potential to incorporate the advantages of layered testbench into current tools equipped with modeling functionalities.

III. Coverage Metrics

The project evaluates two types of coverage metrics, functional coverage and code coverage. Functional coverage is a subjective metric and always defined by the verifier based on a specific verification plan. In this sense, if a verifier drafts an incomplete verification plan, even 100% functional coverage can not gaurantee a fully examined DUT. Although the definition of functional coverage is the same for both directed testing and random verification, the two methods use different ways to gather the coverage information. Code coverage is a metric commonly used with functioanl coverage together to yield a more comprehensive view on the verification process. Similar to functional coverage, 100% code coverage does not mean a correctly functional design either. For example, if one required feature is not implemented in the current DUT, the design might achieve 100% code coverage, but still does not meet the specifications because of the dismissed feature.

A. Functional Coverage

As mentioned above, directed testing and random verification measures functional coverage in different ways. In directed verification, after a targeted test case is verified, the weighted percentage of the verified test case among all required scenarios is added to the overall functional coverage measurement. In random verification, the verifier can only identify which scenarios have been covered by the current testbench after random inputs are generated. In this context, functional coverage is defined as the percentage of expected cases hit by a set of random stimuli. The floating-point adder demo shows the use of functional coverage measurements for random verification.

Table 1 lists the required scenarios for verifying the floating-point adder. The goal of the simple verification plan is not to fully verify the functionality of the design, but to demonstrate how to gather and interpret functional coverage measurements in random verification.

TABLE I. Required Scenarios to Verify Floating-Point Adder

Case	a	b
1	Positive	Positive
2	Negative	Negative
3	Positive with Large Magnitude	Negative with Small Magnitude
4	Positive with Small Magnitude	Negative with Large Magnitude
5	Negative with Small Magnitude	Positive with Large Magnitude
6	Negative with Large Magnitude	Positive with Small Magnitude

In order to collect coverage data, a verifier needs to define covergroups and coverpoints in the testbench. As the name implies, a covergroup groups signals that together target one or more test cases and report the coverage as a unit. For the floating-point adder demo, one covergroup *fpu_coverag* is greated as shown in Fig. 2*,* targeting all six test cases in Table 1. Within the covergorup, there are four individual coverpoints and one cross coverpoint *sign_mag*. Coverpoint *a_sign* and *b_sign* keep track of the signs of the two floating-point numbers and coverpoint *a_mag* and *b_mag* of the magnitudes. The key coverage measurement in this covergroup is the cross coverpoint *sign_mag*, which monitors all combinations of the four individual coverpoint. The 16 combinations of *a_sign, b_sign, a_mag, b_mag* are sufficient to cover all six test cases in Table 1. If one set of random input patterns fall in all combinations, the functional coverage in this case is 100%, meaning that this set of random stimuli have covered all targeted scenarios. Otherwise, there might be test cases missed with this set of random stimuli.

```
// Covergroup Definition
covergroup fpu_coverage @ (posedge clk);
option.goal = 100;

 a_sign : coverpoint my_ifc.a[63] {
  bins a_neg = {1};
  bins a_pos = {0};
 }
 b_sign : coverpoint my_ifc.b[63] {
  bins b_neg = {1};
  bins b_pos = {0};
 }

 a_mag : coverpoint my_ifc.a[62:52] {
  bins a_small = {[0:1024]};
  bins a_big = {[1024:$]};
 }
 b_mag : coverpoint my_ifc.b[62:52] {
  bins a_small = {[0:1024]};
  bins a_big = {[1024:$]};
 }

// Cross Coverpoint Definition
 sign_mag : cross a_sign, b_sign, a_mag, b_mag;

endgroup: fpu_coverage
```

Fig. 2. Covergroup and Cross Coverpoint to Verify Floating-Point Adder

B. Code Coverage

The project uses the ICC tool from Cadence to collect the code coverage information at the same time of runing the simulations. The ICC tool splits the overall code coverage into three categories, block coverage, expression coverage, and toggle coverage. *Block coverage* measures the percentage of a code block that is examined by a testbench. A block is a statement or a sequence of statements in Verilog or VHDl that executes with no branches or delays. One simple example is a sequence of statements between *begin* and *end* keywords without *if* or *case* conditions. *Expression coverage* measures how throughly a testbench exercises expressions in assignments and procedural control constructs, such as *if* or *case* conditions. At last, *toggle coverage* provides information

on untoggled signals or signals that remain constant during a simulation run. With code coverage measurements and detailed reports, a verifier can tailor the constraints on random input generation to cover the overlooked senarios.

IV. RTL FAULT COVERAGE

RTL design and fucntional verification are two steps early in the design flow of integrated circuits. Idenfying and fixing problems at these early stages save 10X time and cost than modiying the design at later stages. The key motivation for RTL fault coverage is to take the advantage of early stages to estimate the gate-level fault coverage of a set of input patterns for a specific fault model and to use the estimation to evaluate the RTL testability and the effectiveness of input patterns generated by ATPG tools.

Previous works on RTL fault coverage include fault models at RTL as in [2] and [3], the RTL fault simulation flow, and the relationship between RTL fault coverage and gate-level fault coverage. For the fault model, the project adopts from [2] the single stuck-at fault for each bit of all variables. To create faulty designs, the original RTL design is modified by inserting a zero-delay buffer between each variable and its executable statements. When no faults are injected, the zero-delay buffer does not affect the logic behavior or the timing of the design. When faults are injected, one of the buffer sets its output to either 0 or 1 regardless of its input. Two modes of fault coverage estimation are available in the RTL fault simulation flow as presented in [2]. Optimistic mode assumes that an injected fault will affect all executable statements which use the variable, while pessimistic mode assumes that the fault will only affect one of the statements involving the variable. Results in [2] show that optimistic estimation and pessimistic estimation both exhibit a similar trend to the actual gate-level fault coverage versus the number of test patterns. The two modes of estimation also set an upper and lower boundary of the actual fault coverage, respectively.

The project follows the fault simulation flow in [2] for the floating-point adder demo. One stuck-at fault is injected at a time. If all N input patterns fail to detect the fault, this set of test vectors miss the specific fault injected. The final RTL fault coverage for the set of N input patterns is calculated as the number of faults detected divided by the total number of all possible faults in the RTL design. In the case of the floating-point adder, the total number of faults is 986. Fig. 3 shows the RTL fault coverage in the optimistic mode for the floating-point adder.

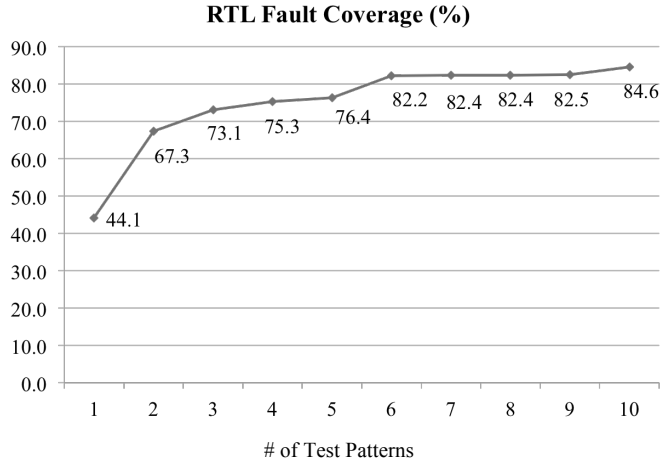

Fig. 3. RTL Fault Coverage for Floating-Point Adder

V. CONCLUSIONS

The goal of the project is to examine the advantages of random verification with real examples. The project focuses on two aspects of random verification, layered testbench and coverage metrics. Object-oriented testbench is an essential part of random verification. Compared with testbench in directed testing, the hierachical testbench has more resusable verification classes. Also, once the testbench is built, a verifier can easily modify the components at the highest level of abstraction in order to improve coverage, saving time and avoiding errors from changing the whole testbench as in the case of directed testing. In addition, the project evalueates two coverage metrics, funcaional coverage and block coverage. The definitions of the two metrics are the same for both directed and random verification, but the ways to measure the coverage information differ for the two methods. Effectively using random stimuli and functional coverage measurements for random verification requires a verifier to think beyond the scope of directed testing, that is not only to think about the necessary test cases to cover the required scenarios, but also to constantly analyze the coverage data and adjust randomization constraints. At last, the project looks into how to use RTL fault coverage to predict gate-level fault coverage. Although there is no one-to-one mapping between the fault coverages at two levels, RTL fault coverage in optimistic mode and pessimistic mode sets an uppder and lower boundary of the gate-level coverage. A verifier can use the RTL estimation to evaluate the RTL testability and the effectiveness of input patterns. Identifying and fixing problems at RTL, an early stage in the design flow, can reduce the overall design time and cost.

REFERENCES

[1] C. Spear and G. Tumbush, *SystemVerilog for Verification.* New York, NY: Springer 2012

[2] Mao, W.; Gulati, R.K., "Improving gate level fault coverage by RTL fault grading," *Test Conference, 1996. Proceedings., International* , vol., no., pp.150,159, 20-25 Oct 1996

[3] Karunaratne, M.; Sagahayroon, A.; Prodhuturi, S., "RTL fault modeling," *Circuits and Systems, 2005. 48th Midwest Symposium on* , vol., no., pp.1717,1720 Vol. 2, 7-10 Aug. 2005